Design Patterns

Definition:

Design patterns are reusable solutions to common problems encountered in software design. They represent best practices evolved over time by experienced developers to address recurring challenges in software development.

Purpose:

1. Abstraction of Solutions: Design patterns provide abstracted solutions to specific design problems, promoting code organization and maintainability.
2. Code Reusability: By encapsulating proven design approaches, patterns enable code reuse, reducing redundancy and promoting efficiency.
3. Communication: Design patterns serve as a common language among developers, facilitating effective communication and understanding of software structures.

Categories:

Design patterns are typically categorized into three main types:

1. **Creational Patterns**: Concerned with object creation mechanisms, such as Singleton, Factory Method, and Abstract Factory.
2. **Structural Patterns**: Focus on object composition, including patterns like Adapter, Decorator, and Proxy.
3. **Behavioral Patterns**: Address communication between objects, featuring patterns like Observer, Strategy, and Command.

Benefits:

1. **Scalability**: Design patterns contribute to scalable and maintainable software architectures, allowing systems to evolve with changing requirements.
2. **Flexibility**: They enhance flexibility by providing adaptable solutions to specific problem domains, ensuring that software can be modified and extended.
3. **Efficiency**: By leveraging proven solutions, design patterns promote efficient development, reducing the likelihood of errors and promoting code quality.

Singleton Pattern

Definition:

The Singleton Design Pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. It restricts the instantiation of a class to a single object, offering a single, globally accessible point to that instance.

Key Characteristics:

1. **Single Instance**: Ensures that only one instance of the class exists in the entire application.
2. **Global Access Point**: Provides a global point of access to the single instance, allowing it to be easily accessible from any part of the code.
3. **Lazy Initialization:** The instance is created only if it is requested for the first time, allowing for lazy initialization.
4. **Private Constructor**: The class typically has a private constructor to prevent direct instantiation from external classes.
5. **Thread Safety**: Implementation should handle concurrency issues to ensure thread safety, especially in a multithreaded environment.

Use Cases:

1. Resource Management: When a single instance is required to manage resources such as database connections, file systems, or network connections.
2. Configuration Settings: Singleton pattern is suitable for managing configuration settings or application parameters.
3. Logging: To control access to a shared resource like a log file, ensuring only one logger instance exists.
4. Caching Mechanisms: In scenarios where a single instance is responsible for caching frequently used data.

Implementation:

1. Private Constructor: Ensure the class has a private constructor to prevent external instantiation.

```
class Singleton:
    _instance = None

    def __new__(cls):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance
```

2. Lazy Initialization: Create the instance only when it is accessed for the first time.

```
class Singleton:
    _instance = None

    def __new__(cls):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance
```

3. Thread Safety: Implement thread-safe creation of the singleton instance.

```
import threading

class Singleton:
    _instance = None
    _lock = threading.Lock()

    def __new__(cls):
        with cls._lock:
            if not cls._instance:
                cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance
```

Pros:

1. Global Access: Provides a single point of access to a shared instance.
2. Resource Efficiency: Reduces memory usage by having only one instance.
3. Lazy Initialization: Efficient use of resources by creating the instance on-demand.

Cons:

1. Global State: Can introduce a global state, making the application less modular.
2. Testing Challenges: Testing can become challenging due to the global state.

Conclusion:

The Singleton Design Pattern is a powerful pattern when there is a need for a single, globally accessible instance. While it should be used judiciously, it provides an efficient way to manage shared resources and control access to critical components within an application.

Prototype Design Pattern

Definition:

The Prototype Design Pattern is a creational design pattern that focuses on creating objects by copying an existing object, known as the prototype. Instead of creating new instances using traditional constructors, the pattern allows objects to be cloned. This approach is particularly useful in situations where object creation is resource-intensive or complex.

Key Characteristics:

Cloning Mechanism:
- Objects are created by copying an existing instance (prototype) rather than using constructors.

Prototype Interface:
- Defines a common interface or abstract class that declares a method for cloning.

Concrete Prototypes:
- Classes that implement the prototype interface and provide the actual implementation of the clone method.

Client:
- Initiates the cloning process by requesting the prototype to clone itself.

Shallow and Deep Copy:
- Allows for both shallow copy (copying top-level attributes) and deep copy (recursively copying nested objects).

Use Cases:

Efficient Object Creation:
- When object creation is more efficient by copying an existing object rather than initializing a new one from scratch.

Reducing Subclassing:
- In scenarios where a class cannot anticipate the type of objects it must create, prototype pattern allows for new object creation without relying on subclasses.

Configuring Objects:
- Useful for configuring complex objects with different properties.

Implementation:

Prototype Interface:
- Define an interface or abstract class with a clone method.

```
from abc import ABC, abstractmethod
```

```
class Prototype(ABC):
 @abstractmethod
 def clone(self):
 pass
```

Concrete Prototypes:

- Implement the clone method in concrete classes.

```
class ConcretePrototypeA(Prototype):
 def clone(self):
 return ConcretePrototypeA()

class ConcretePrototypeB(Prototype):
 def clone(self):
 return ConcretePrototypeB()
```

Client:

- Request cloning from the prototype.

```
class Client:
 def create_instance(self, prototype):
 return prototype.clone()
```

Pros:

Object Creation Efficiency:

- Efficiently create new objects by copying existing ones.

Flexible Object Creation:

- Create objects with different configurations or states based on existing prototypes.

Reduced Complexity:

- Simplifies client code by abstracting the object creation process.

Maintains Object Relationships:

- Helps maintain relationships between objects, especially when not trivial to establish.

Cons:

Cloning Complexity:

- Implementing the cloning logic for objects can be complex, particularly with deep copies or circular references.

Initialization Challenges:

- Cloned objects may require additional initialization steps after cloning to be in a valid state.

Conclusion:

The Prototype Design Pattern provides an efficient way to create objects by copying existing ones, offering flexibility and reduced resource usage. While it requires careful handling of object cloning, it is valuable in scenarios where object creation is resource-intensive or where variations of objects need to be created without subclassing.

Command Pattern

The command pattern is a data-driven design pattern and falls under the behavioral pattern category. A request is wrapped under an object as a command and passed to the invoker object. The invoker object looks for the appropriate object that can handle this command and passes the command to the corresponding object which executes the command.

The Command design pattern encapsulates a request as an object, allowing users to parameterize clients with queues, requests, and operations. It allows for the separation of concerns between an object that invokes an operation, called the client, and the object that operates, called the receiver.

History:

The Command Pattern has been around for a long time and is deeply rooted in object-oriented design principles. It became widely known after being described in the famous book "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (commonly known as the Gang of Four, or GoF)

Real World Example:

Objects in Command:

- Command
    - Sends command to receiver
- Receiver
    - Performs command task
- Invoker
    - Invokes command, bookeeps execution
- Client
    - Contains command and invoker objects, sends command object to invoker

Advantages:

The main advantage of the command pattern is that the invoker and receiver are low coupled objects. This means that any changes to one will not directly affect the other. Another advantage of using design patterns in general is good organization and readability in code.

Disadvantages:

Disadvantage of Command The main drawback is that Command requires more objects to invoke a command than just a single method object.

Why & How to Implement:

If you find your code using a lot of if-else statements to branch into different functions, consider using the Command Pattern. Keys to implementing: Separate your code into an invoker, command objects, and receiver so that the code is low coupled and more readable under a client function.

Abstract Factory Pattern

Definition:

The Abstract Factory Design Pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. It encapsulates a group of individual factories (concrete factories) that share a common theme, allowing clients to create families of objects without specifying their concrete classes.

Key Characteristics:

Abstract Factory Interface:
- Declares a set of methods to create abstract product objects.

Concrete Factories:
- Implement the abstract factory interface, producing families of related products.

Abstract Products:
- Declare interfaces for a set of distinct products, each produced by a family of concrete factories.

Concrete Products:
- Implement the interfaces defined by abstract products.

Client:
- Uses the abstract factory and abstract product interfaces to create families of related objects.

Use Cases:

Dependency Management:
- Ideal for managing dependencies between multiple interrelated objects.

Configurable Systems:
- Useful in systems that need to be configured with multiple families of related objects.

Product Variants:
- When there are multiple variants of a product family, and the system needs to be configured with one of them.

Implementation:

Abstract Factory Interface:
- Declare a set of creation methods for each type of abstract product.

```
from abc import ABC, abstractmethod

class AbstractFactory(ABC):
@abstractmethod
def create_product_a(self):
pass

@abstractmethod
def create_product_b(self):
pass
```

Concrete Factories:

- Implement the abstract factory interface to create families of related products.

```
class ConcreteFactory1(AbstractFactory):
        def create_product_a(self):
                return ConcreteProductA1()

        def create_product_b(self):
                return ConcreteProductB1()

        class ConcreteFactory2(AbstractFactory):
        def create_product_a(self):
                return ConcreteProductA2()

        def create_product_b(self):
        return ConcreteProductB2()
```

Abstract Products:

- Declare interfaces for the products created by the abstract factory.

```
class AbstractProductA(ABC):
        @abstractmethod
        def operation_a(self):
                pass

class AbstractProductB(ABC):
        @abstractmethod
        def operation_b(self):
        pass
```

Concrete Products:

- Implement the interfaces defined by abstract products.

```
class ConcreteProductA1(AbstractProductA):
        def operation_a(self):
                return "ConcreteProductA1 Operation"

class ConcreteProductB1(AbstractProductB):
        def operation_b(self):
                return "ConcreteProductB1 Operation"
```

Pros:

Product Consistency:

- Ensures that created objects are compatible and belong to the same family.

Flexibility:

- Allows easy switching between different families of products.

Encapsulation:

- Encapsulates product creation, making it easier to introduce new product families.

Cons:

Complexity:

- The pattern may introduce additional complexity when dealing with a large number of product families.

Conclusion:

The Abstract Factory Design Pattern provides an interface for creating families of related or dependent objects, promoting consistency, and making it easier to adapt the system to different variants. It is particularly valuable in scenarios where the creation of multiple, interrelated objects needs to be managed in a cohesive manner.

Bridge Pattern

Definition

The Bridge Design Pattern divides and organizes a single class that has multiple variants of some functionality into two hierarchies: abstractions and implementations. By doing this, the client code won't be exposed to implementation details as it will only work with high level abstractions.

History

The bridge design pattern is one of the 23 design patterns introduced by the Gang of Four, composed of Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, on a book called "Design Patterns: Elements of Reusable Object-Oriented Software", published on October 21, 1994.

Usages

- Decoupling Interface and Implementation
    - Separate high-level logic from low-level operations.
    - Independent evolution of both parts.
- Platform Independence
    - Operate across multiple platforms/environments.
    - Example: Different graphics rendering engines (DirectX, OpenGL).
- Extensibility
    - Extend main logic and operations without mutual impact.
    - Add features without altering core functionality.
- Dynamic Binding
    - Determine implementation at runtime.
    - Useful for plugin architectures and dynamic module loading.

Real-world examples

- The Bridge pattern allows e-commerce platforms to integrate multiple payment gateways without changing the checkout interface.
- The Bridge Design Pattern is applicable to music player apps that handle many file formats. The Bridge pattern is used to separate the abstraction (the music player interface) from the implementation (file format handling) in this classic example.

Pros

- **Cleaner Code**: Separation of concerns leads to more organized and readable code.
- **Maintainability**: Changes in implementation don't affect the client code.
- **Scalability**: Easily introduce new implementations without major code
- **Flexibility**: Mix and match different implementations as needed.

Cons

- **Complexity**: When abstractions and implementations increase, the Bridge pattern might complicate a program. Code complexity can make it difficult to comprehend and maintain.
- **Multiple Indirections**: Several indirections within an application has the potential to adversely impact its performance. This can be attributed to the inherent indirection that occurs when a request is passed from the Abstraction to the Implementor.

Composite Design Pattern

What is composite pattern?

- is a partitioning design pattern and describes a group of objects that is treated the same way as a single instance of the same type of object?
- should be used when clients need to ignore the difference between compositions of objects and individual objects.

History

This pattern was first introduced by the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) in their book "Design Patterns: Elements of Reusable Object-Oriented Software," which was published in 1994.

Pros

- You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
- Open/Closed Principle. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

Cons

- It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

Real Life Examples

- Company
- Menu Systems

Four participants

- Component - declares the interface for objects in the composition and for accessing and managing its child components.
- Leaf - defines behavior for primitive objects in the composition.
- Composite - stores child components and implements child related operations in the component interface.
- Client - manipulates the objects in the composition through the component interface.